submission to IEEE Computer Graphics and Applications (2009)

Wait-Free Shared-Memory Irradiance Caching

Kurt Debattista[1], Piotr Dubla[1], Luís Paulo Santos[2] and Alan Chalmers[1]

[1]International Digital Laboratory, WMG, University of Warwick, United Kingdom
[2]Departamento de Informática, Universidade do Minho, Portugal

Abstract

Parallelizing rendering algorithms to take advantage of multicore machines is not a straightforward task. Certain methods require frequent synchronization among threads to obtain benefits similar to the sequential algorithm. One such algorithm is the Irradiance Cache (IC), an acceleration data structure which caches indirect diffuse irradiance values. In multicore systems the IC must be shared among threads in order to achieve high efficiency levels. We propose a novel wait-free access mechanism to the shared IC, which does not make any use of the commonly used blocking or busy waiting methods, avoiding most serialization and reducing contention. We compare with two classical approaches: a lock based mechanism and a local write technique. We demonstrate, using two systems with up to 24 cores, that the wait-free approach significantly reduces synchronization overheads, thus resulting in improved performance.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.7]: Three Dimensional Graphics and Realism—Ray Tracing Computer Graphics [I.3.1]: Hardware Architecture—Parallel Processing

1. Introduction

The advent of multicore computing to the desktop has meant that in order to maximize the use of available resources, traditional rendering algorithms need to be modified to account for the parallelism. While, for certain algorithms, such as classical ray tracing, this conversion can be relatively straightforward, when computing more complex lighting conditions, methods which share data among multiple parallel threads do not afford the same opportunities. Under such conditions careful consideration must be paid to synchronization among threads to ensure that overheads are kept to a minimum and computation can proceed unhindered [Her09]. In this paper we present a parallel solution to one such method, the irradiance cache (IC). The IC offers an interesting challenge for shared memory processing due to the frequent access of a shared data structure by multiple threads. We demonstrate how to parallelize this algorithm using a wait-free approach, which may inspire other solutions to similar problems in computer graphics.

Rendering global illumination light transport effects within a ray tracing context is a computationally demanding task. Recent improvements in the field of ray tracing have made it possible to interactively compute many of the global effects, such as specular phenomena and correct shadows [WMG*07]. Indirect diffuse interreflections, however, require dense sampling of the hemisphere at each shading point, dramatically increasing rendering times. Ward et al. [War88] exploited the fact that the indirect diffuse component is generally a continuous smooth function over space not affected by the high frequency changes common with the specular component. They proposed the IC data structure to allow sparse evaluation of indirect diffuse irradiance. Sparsely calculated irradiance values are stored in the IC and reused to extra(inter)polate irradiance values at nearby locations. By exploiting spatial coherence, the IC can achieve an order of magnitude improvement in rendering time over unbiased Monte Carlo integration.

In multithreaded shared memory systems the IC must be shared to avoid replicated computations of diffuse samples among rendering threads, thus increasing efficiency. Since all rendering threads can write to and read from the IC, a data access control mechanism is required to ensure that the data structure is not corrupted. Such control mechanisms incur overheads, such as serialization of accesses to the shared data structure; it must thus be carefully designed in order not to compromise performance. Traditionally, access to shared memory data structures is controlled via lock-based mutual exclusion techniques. However, there are alternatives termed

non-blocking data structures. Non-blocking data structures can take the form of obstruction-free, lock-free or wait-free data structures, that offer considerable performance advantages. The most powerful of these are the wait-free methods. For further details see Section (Box) 3. In reality, practical wait-free data structures are considered difficult to construct and are relatively rare [HLM03].

In this paper we propose an efficient wait-free algorithm which allows all threads to concurrently access an unbounded shared IC, without using any locks or critical sections. We take this approach and compare it with two other mechanisms which share the IC among threads on a shared memory system. The first is based on traditional locking techniques and locks the shared IC every time a thread accesses it, both for reading and writing. The second is a local copy method which avoids concurrent access control by maintaining a local IC, per thread, and merging at the end of each frame. Efficient sharing of the IC in multithreaded systems is mandatory in order to achieve high efficiency levels, since computed irradiance values become readily available to all threads, thus avoiding work replication. This is especially relevant because utilization of the IC has increased significantly over the last few years and has been used as an inspiration for a myriad of new caching algorithms which could benefit from our approach, see Box (Section) 2 for further details.

This paper's contributions are the proposal of an efficient wait-free algorithm for sharing the IC among rendering threads on shared memory systems and a comparison of the proposed algorithm's efficiency with two traditional data access control mechanisms: a lock-based approach and a local copy one. This is an extension of our previous work [DDSC09], where we presented an initial version of the wait-free algorithm and tested it on an eight core machine. The previous wait-free algorithm could only handle fixed sized arrays in the IC structure. Furthermore, the algorithm caused some data to be discarded when a conflict among threads occurred. These limitations have been fixed in the current work by handling dynamically allocated memory, thus supporting an unbounded number of cache samples, and guaranteing the successful insertion of all new irradiance values. We assess this algorithm's efficiency on two highly concurrent multicore systems with up to 24 physical cores and show that it exhibits superior performance and scalability properties than the lock-based and local cache alternatives.

This paper is structured as follows. Section (Box) 2 presents the motivation and reasoning behind the irradiance cache, while Section (Box) 3 briefly discusses non-blocking synchronization. In section 4 we present related work and in Section 5, we present the algorithms for the three data access control mechanisms. Section 6 compares results and, finally, in Section 7 we conclude and describe possible future work.

2. Irradiance Caching (Box)

Physically-based computation of the radiance reflected at a point p along a direction Θ is dictated by the rendering equation [Kaj86]:

$$L_r(p \rightarrow \Theta) = \int_{\Omega} f_r(p, \Theta \leftrightarrow \Psi) L_i(p \leftarrow \Psi) \cos(\overrightarrow{N_p}, \Psi) d_{\Omega_\Psi}$$

where $f_r(p, \Theta \leftrightarrow \Psi)$ is the BRDF at point p for the pair of directions Θ and Ψ, $L_i(p \leftarrow \Psi)$ is the incident radiance at p along direction Ψ, $\overrightarrow{N_p}$ is the surface normal at p and Ω, the integration domain, is the hemisphere centered at p and oriented around $\overrightarrow{N_p}$.

Ray tracing approximates $L_r(p \rightarrow \Theta)$ by shooting rays along a number of directions Ψ, thus sampling $L_i(p \leftarrow \Psi)$ for these directions. Certain light transport phenomena, such as specular scattering of light and direct illumination, can usually be computed with a limited number of rays; other phenomena, due to their lack of directionality, require hemisphere sampling, i.e., stochastically selecting and shooting a large number of rays across Ω – this is a stochastic integration method known as Monte Carlo integration, which usually accounts for a large amount of the computation.

One such phenomenon is indirect diffuse reflection, defined as diffusely reflected radiance at point p along a given direction due to indirect irradiance $E(p)$. $E(p)$ is the indirect incident radiant flux per unit area at p. Accurately computing $E(p)$ requires densely sampling Ω, which in a ray tracing context is achieved by shooting hundreds or thousands of rays distributed across this hemisphere (while carefully avoiding directions corresponding to light sources, since we only want to include indirect lighting, i.e., light that has been reflected by at least one object).

Indirect diffuse reflections are crucial to convey a perception of realism (See Figure 1), but sampling the hemisphere at all shading points results in very long rendering times, deemed unacceptable even for most offline renderings. Ward et al. [War88] realized back in 1988 that indirect diffuse reflection is generally a continuous smooth function over space, not affected by the high frequency changes common with specular reflections. They proposed accelerating the computation of indirect diffuse reflections by densely sampling the hemisphere at a sparse set of shading points only and interpolating for the remaining ones. Sparsely calculated indirect irradiance values are stored in the irradiance cache (IC) data structure and later reused to extra(inter)polate irradiance values at nearby locations.

Indirect irradiance at p, $E(p)$, can be interpolated from a set $S(p)$ of previously evaluated irradiance values $E(p_i)$ at points p_i, by using a weighted average:

$$E(p) = \frac{\sum_{i \in S(p)} w_i(p) E(p_i)}{\sum_{i \in S(p)} w_i(p)}$$

where the weights $w_i(p) = (\frac{\|p - p_i\|}{R_i} + \sqrt{1 - \overrightarrow{N_p} \cdot \overrightarrow{N_{p_i}}})^{-1}$ depend on the distance between p and p_i, on the harmonic

(a) Direct only (b) Indirect only (c) Full

Figure 1: The contribution of indirect lighting

mean distance to objects visible from p_i, R_i, and on the relative orientation of the normals at p and p_i. The set $S(p)$ of points p_i that can be used to interpolate $E(p)$ is determined by requiring that the weights $w_i(p)$ are larger than the reciprocal of the maximum acceptable error, a, which is a user supplied parameter: $S(p) = \{i : w_i(p) > 1/a\}$. When indirect irradiance at any point p is required the renderer first determines the set $S(p)$ by querying the IC. If $S(p)$ is empty, $E(p)$ is evaluated by Monte Carlo integration, otherwise it is interpolated from the $E(p_i)$ belonging to $S(p)$. Querying the IC to determine $S(p)$ amounts to locating all samples p_i stored in the cache that satisfy the above criterium; this is referred to as a range search, a computationally demanding task that can be optimized by spatially ordering the IC by resorting to 3D hierarchical data structures, such as octrees or kd-trees.

By exploiting spatial coherence, the IC offers an order of magnitude improvement in rendering time over Monte Carlo integration. Performance is further improved when rendering animations of static scenes, since the indirect diffuse irradiance remains constant and the IC samples can thus be reused across frames.

The IC has been extended as a stand-alone algorithm in many guises recently: as an acceleration data structure for rendering glossy surfaces by storing radiance [KGPB05], participating media phenomena [JDZJ08] and translucency [KLC06], or used in conjunction with photon mapping [Jen01]. It has been extended to exploit coherence in the temporal domain [SKDM05, GBP07, DDB*09]. Similar methods have also been used to accelerate the rendering in production renderers at PDI/Dreamworks [TL04].

References

[DDB*09] DEBATTISTA K., DUBLA P., BANTERLE F., SANTOS L. P., CHALMERS A.: Instant caching for interactive global illumination. *Comput. Graph. Forum 28*, 8 (2009), 2216–2228.

[GBP07] GAUTRON P., BOUATOUCH K., PATTANAIK S.: Temporal radiance caching. *IEEE Transactions on Visualization and Computer Graphics 13*, 5 (2007), 891–901.

[JDZJ08] JAROSZ, DONNER, ZWICKER, JENSEN: Radiance caching for participating media. *ACM Trans. on Computer Graphics 27*, 1 (March 2008).

[Jen01] JENSEN H. W.: *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., 2001.

[Kaj86] KAJIYA J. T.: The rendering equation. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1986), ACM, pp. 143–150.

[KGPB05] KRIVANEK J., GAUTRON P., PATTANAIK S., BOUATOUCH K.: Radiance caching for efficient global illumination computation. *IEEE Trans. on Visualization and Computer Graphics 11*, 5 (2005), 550–561.

[KLC06] KENG S.-L., LEE W.-Y., CHUANG J.-H.: An efficient caching-based rendering of translucent materials. *Vis. Comput. 23*, 1 (2006), 59–69.

[SKDM05] SMYK M., KINUWAKI S., DURIKOVIC R., MYSKOWSKI K.: Temporally Coherent Irradiance Caching for High Quality Animation Rendering. *Computer Graphics Forum 24*, 3 (2005), 401–412.

[TL04] TABELLION E., LAMORLETTE A.: An Approximate Global Illumination System for Computer Generated Films. *ACM Trans. on Graphics 23*, 3 (2004), 469–476.

[War88] WARD G.: A ray tracing solution for diffuse interreflection. *Computer Graphics - SIGGRAPH'88 22*, 4 (August 1988).

End Irradiance Caching Box

---

3. Non-blocking Synchronization (Box)

When using shared memory multithreading to execute parallel computations, care must be taken when accessing data structures that are accessible to all the threads concurrently. When multiple threads can read and write from a shared data structure, a data access control mechanism is required to ensure that the data structure is not corrupted.

Traditionally, access control to shared memory data structures is maintained via mutual exclusion. Blocking mechanisms, that preempt the running thread, may be used when critical sections are reasonably large. Alternatively, when

frequent access to a shared data structure may be required, the cost of blocking may be prohibitive. In such case a thread is made to busy wait, usually using a spin lock, if another thread lies within the critical section, until access is allowed. Such control mechanisms incur overheads, such as serialization of accesses to the shared data structure. Blocking entails expensive context switches and busy waiting of frequently-accessed resources leads to contention which can drastically reduce performance as the number of threads increases [ALL89].

Alternatives that avoid mutual exclusion do exist in the form of non-blocking synchronization. By carefully ordering instructions, non-blocking algorithms can guarantee that none of the code is serialized, due to the removal of all critical sections, resulting in a reduction in contention [HS08]. The weakest form of non-blocking data structures take the form of obstruction-free methods that guarantee that a thread can complete in finite time if it operates in isolation. When non-blocking data structures can guarantee that at least one among a set of concurrent threads will complete in finite time, they are said to be lock free. All lock free algorithms are obstruction free. Lock-free and obstruction free methods rely on retrials and cannot guarantee an upper bound on the number of executed instructions. When all threads are guaranteed to complete in finite time the algorithm is said to be wait free. Wait-free algorithms can guarantee an upper bound on the number of instructions, thus avoiding starvation, deadlock and livelock, priority inversion problems and are ideal for multiprogrammed multiprocessors, for example when a thread holding a lock is preempted causing all other threads to busy wait uselessly. Clearly all wait-free data structures are also lock free.

The construction of non-blocking algorithms requires the use of powerful atomic primitives which are executed as a single instruction, without any interruption, on modern architectures. These algorithms can be seen as a limiting case on the reduction of the size of critical sections, reducing them to these individual machine instructions. We show pseudo code (Listings 1 and 2) for the two atomic instructions, compare and swap (CAS) and fetch and add (XADD), that we will be using for our wait-free IC. Herlihy [Her91] provides a hierarchy of the effectiveness of such primitives, the most effective being those that can be used to implement any wait-free data structure which he described as being compare and swap (or load-link store-conditional instruction pair, which are an alternative to compare and swap found on some architectures).

Listing 1: Fetch and Add

```
1 atomic XADD(address location)
2 {
3     int value = *location;
4     *location = value + 1;
5     return value
6 }
```

Listing 2: Compare and swap

```
1 atomic CAS(address location, value cmpVal, value newVal)
2 {
3     if (*location == cmpVal) {
4         *location = newVal;
5         return true;
6     } else return false;
7 }
```

References

[ALL89] ANDERSON T. E., LAZOWSKA E. D., LEVY H. M.: The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Trans. Computers 38*, 12 (1989), 1631–1644.

[Her91] HERLIHY M.: Wait-free synchronization. *ACM Trans. Program. Lang. Syst. 13*, 1 (1991), 124–149.

[HS08] HERLIHY M., SHAVIT N.: *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

End Non-blocking Synchronization Box

---

4. Related Work

The IC is an acceleration data structure which caches indirect diffuse irradiance samples, within the framework of a distributed ray-tracing algorithm [War88]. Irradiance values can then be interpolated for regions within a given neighborhood of these samples, thus reducing rendering time by exploiting spatial coherence. Section (Box) 2 describes the motivation and mechanism behind this data structure.

In order to accelerate range searches, performed to locate valid samples within the IC, an octree is incrementally built every time a new sample is added; writing to the cache requires both storing the new indirect diffuse irradiance value and updating the octree topology. In parallel systems each rendering process, or thread, might evaluate new indirect diffuse irradiance values and add them to the IC. In order to increase efficiency, the IC must be shared among all processes, thus avoiding replicated work, where one process evaluates an irradiance value that could have been interpolated from irradiance values evaluated by other processes. The IC becomes a shared data structure, thus requiring some sharing mechanism assuring that all processes can access the available data, that the data is not corrupted and that overheads do not compromise efficiency.

In distributed memory systems, such as clusters of workstations, each node has its own address space, resulting in multiple copies of the shared data structure that are regularly synchronized. The standard Radiance distribution [War94] supports a parallel renderer over a distributed system using the Network File System for concurrent access of the IC; this has been known to lead to contention and may result

in poor performance when using inefficient file lock managers. Koholka et al. [KMG99] broadcast IC values amongst processors after every 50 samples calculated at each slave. Robertson et al. [RCLL99] presented a centralized parallel version of Radiance whereby the calculated IC values are sent to a master process whenever a threshold is met. Each slave then collects the values deposited at the master by the other slaves. [DSC06] propose restricting diffuse irradiance evaluations to a subset of the available processors and synchronizing the IC among these at a higher frequency than with the remaining processors.

We are not aware of any publication describing a data access control mechanism for sharing the irradiance cache among rendering threads in a shared memory parallel system, other than our previous wait-free algorithm [DDSC09]. The algorithm proposed in this paper extends this previous one, supporting extendable memory for inserting an unbounded number of IC samples and also successfully inserting of all new irradiance values, some of which were discarded when a conflict occurred in the previous method. Furthermore, the algorithm is evaluated on two highly concurrent multicore architectures, demonstrating its superior performance and scalability properties.

5. Algorithms

In this section the algorithms for the three evaluated data access control mechanisms are presented. To begin with we show a traditional single-threaded IC with no access control in Listing 3. IrradianceCache is the data structure that represents the IC. It is composed of an octree of recursive nodes. The individual node is termed ICNode. Each ICNode contains pointers to another eight nodes and an ICList storing the list of IC samples. Figure 2 shows the ICNode structure for the wait-free method. For the other methods the ICList is just a single dynamic array. In the subsequent sections we demonstrate how the traditional approach can be modified to enable the different access control algorithms.

Listing 3: Traditional sequential IC

```
1  IrradianceCache IC;
2
3  ComputeIndirectDiffuse() {
4      //get irradiance from IC if there are valid records
5      inIC = IC.getIrradiance ();
6      if (!inIC) { // no valid records found
7          // compute it by sampling the hemisphere
8          ICsample = ComputeIrradianceRT ();
9          // insert new IC sample into the octree
10         IC.insert (ICsample);
11     }
12 }
13
14 IrradianceCache::getIrradiance(Irr) {
15     Irr = {0,0,0};
16     <Traverse the octree>
17     <verify validity of sample>
18     <extrapoloate irradiance; add to Irr>
19     if (found) return true;
20     else return false;
21 }
22
23 IrradianceCache::insert (ICsample) {
24     // recursively traverse the octree
25     // starting at root
26     IC.root.insert (ICsample);
27 }
28
29 ICNode::insert (ICSample) {
30     if (correct insertion node) {
31         IClist.Add (ICsample);
32     } else {
33         // go deeper in the octree
34         xyz = EvaluateOctant();
35         if (children[xyz] == NULL)
36             children[xyz] = new ICNode ();
37         children[xyz].insert (ICsample);
38     }
39 }
40
41 ICList::Add (ICsample) {
42     // insert new record in head of list
43     IClist.records[head++] = ICsample;
44 }
```

5.1. Lock-Based Irradiance Cache (LCK)

The lock-based access control algorithm locks the IC whenever a read or write is made to it (Listing 4 lines 4 - 6, 12 - 14). However, the code responsible for hemisphere sampling, ComputeIrradianceRT(), is not a critical region, thus allowing concurrent evaluation of irradiance. The major disadvantage of the LCK approach is that it serializes all accesses, both reads and writes, to the shared IC. As the number of threads increases, contention will also increase, preventing performance from scaling with the degree of parallelism.

Listing 4: Lock-based IC

```
1  ComputeIndirectDiffuse()
2  {
3      //get irradiance from IC if there are valid records
4      IC.lock();
5      inIC = IC.getIrradiance (Irr);
6      IC.unlock();
7
8      if (!inIC) { // no valid records found
9          // compute it by sampling the hemisphere
10         ICsample = ComputeIrradianceRT ();
11         // insert new IC sample into the octree
12         IC.lock();
13         IC.insert (ICsample);
14         IC.unlock();
15     }
16 }
```

5.2. Local-Write Irradiance Cache (LW)

An alternative approach is to have a global IC readable by all threads and an additional local IC per thread; each thread writes only to its local IC but reads from both. At certain predefined execution points, such as the end of a frame, the local ICs are sequentially merged into the global IC. This form of synchronization uses an end of frame as a barrier, effectively constituting a blocking approach to synchronization.

The major drawback of this approach is that it does not allow for any sharing within a single frame, thus resulting in work replication. The LW algorithm has a much higher IC sample count than the other two approaches, since each thread must locally evaluate all irradiance values required by its assigned image tiles. Additionally, memory consumption is dictated by the number of threads being used and the complexity of the octree itself.

Listing 5: Local-Write IC

```
1  IrradianceCache IClocal[number threads], ICglobal;
2
3  ComputeIndirectDiffuse()
4  {
5      //get irradiance from IC if there are valid records
6      inIC = ICglobal.getIrradiance (Irr);
7
8      if (!inIC)
9          inIC = IClocal[current thread].getIrradiance ();
10
11     if (!inIC) { // no valid records found
12         // compute it by sampling the hemisphere
13         ICsample = ComputeIrradianceRT ();
14         // insert new sample into the local cache
15         IClocal[current thread].insert (ICsample);
16     }
17 }
```

5.3. Wait-Free Irradiance Cache (WF)

The wait-free algorithm does not rely on any critical sections to both read and write to the shared IC. The algorithm makes changes to three methods from the traditional IC.

The first method we describe is the ICList::Add function, see Listing 6. The insert onto the node itself takes the form of an insertion onto an array or an unbounded queue. For the sake of completeness we demonstrate how the method works for an unbounded queue. Insertion onto a fixed sized array is just a specialized case of this algorithm. The version of the fixed size array is the same as the enqueue function of the Herlihy Wing concurrent queue [HW90]. The structure used for ICList is an unbounded queue composed of a queue of arrays, used to maintain coherence and to ensure that the extending of the queue does not occur frequently, see Figure 2. In Listing 6 the array type is denoted as qNode. qNode contains an array of qNodeSize elements and a pointer to another qNode. Initially the queue of arrays contains only one qNode, whenever required a new qNode is created and attached to the previous one. An array is always initialized with a list of NULL pointers (or some other symbol that is not used by the computation) to denote that an ICsample has not yet been added. When adding samples to an IC node the atomic XADD operator (Listing 6 line 3) is used, returning a unique index into the list of records, which ensures that samples are never over-written; simultaneously, the index to the next free position is incremented.

When the structure needs to be extended (Listing 6 line 13) a new qNode is created and a CAS is used to insert it onto the previous qNode (Listing 6 line 19). If the queue has not yet been extended (by another thread), indicated by the pointer still being NULL, then CAS completes successfully and the associated ICsample is inserted onto the structure (Listing 6 line 26). If, however, another thread extended the queue, then CAS will fail and this thread will discard the created qNode (Listing 6 line 20) and insert the associated sample onto the qNode that some other thread must have created (otherwise the CAS would have succeeded).

Figure 3 demonstrates an example of this method being executed by three concurrent threads, R, G, B for a qNodeSize of five. This will help illustrate how the ICList:Add() method functions. At 3a the ICList is completely empty. At 3b, R has just incremented the head but has not yet inserted the sample. R inserts the sample at 3c. At 3d, both B and G have just incremented the head but not inserted the samples. At 3e, B has not inserted its sample and R has inserted another sample, but G is yet to insert the sample and is still on the same line of code that it was on at 3d.

3f demonstrates the scenario when the list needs to be extended and possible conflict may occur. R has just filled in the first qNode and G and B are about to insert another two samples, they have in fact already incremented head. Since both threads G and B have checked that the last qNode is full and has no successor (Line 16), both created a new qNode. However, due to the CAS at Line 19 only one will succeed in attaching it to the previous qNode. In this case, at 3g, we can see that G has succeeded and B is deleting the qNode it created. G has inserted the sample onto the new qNode. At 3h, B inserts the sample onto the qNode that G had created.

Listing 6: Wait-Free IC Add

```
1  ICList::Add (ICsample) {
2      // get index of new sample in node list
3      int index = XADD (&head);
4      int iteration = index / qNodeSize;
5      int pos = index % qNodeSize;
6      qNode * tail;
7      int count = 0;
8
9      // identify node - can be optimized with a local tail
10     for (tail = qHead; tail->next != NULL && count <
           iteration;
11         tail = tail->next, count++);
```

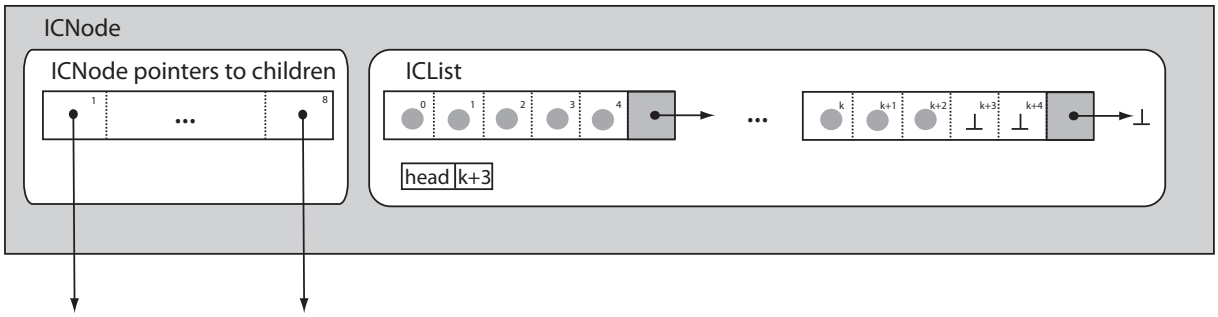


Figure 2: The structure of an ICNode for the wait-free method. For the other methods ICList is just a single dynamic array which is extended whenever required.

```
12
13      if (iteration > count){
14          for (int n = 0; n < iteration − count; n++) {
15                  // this is where we add the new Array
16                  if (tail−>next == NULL) {
17                      // all entries are initialized as NULL
18                      qNode ∗ newN = new qNode;
19                      if(!CAS(&tail−>next, NULL, newN))
20                          delete newN;
21                  }
22                  tail = tail−>next;
23      }
24      // if this thread did not update
25      // some other thread must have updated
26      tail−>records[pos] = ICsample;
27      return index;
28  }
```

The second method that is changed from the traditional IC is the insert onto the octree structure, see Listing 7. When adding a new child node to the octree, the new node is built using a temporary pointer. Once built, the node is attached to the octree using the CAS operator (Listing 7 line 10). The reasoning used for this method is similar to that used for ICList::Add. Either this thread creates the subtree or some other thread does and computation proceeds notwithstanding.

Listing 7: Wait-Free IC insert

```
1   ICNode::insert (ICSample) {
2       if (correct insertion node)
3           IClist.Add (ICsample);
4       else { // go deeper in the octree
5
6           xyz = EvaluateOctant();
7           if (children[xyz]==NULL) {
8               temp = new ICNode();
9               // Update new branch into the octree
10              if (!CAS (children[xyz], NULL, temp))
11                  free temp;
12          }
13          // irelevant to whether this thread created the subtree
14          // someone must have created anyway
15          // recurse the insertion of ICsample onto the subtree
16          children[xyz].insert (ICsample);
17      }
18  }
```

The final method that is modified is IrradianceCache::getIrradiance(), see Listings 8. The modifications to this function are just there to reflect the change in structure to ICList. This method makes use of the fact that qNode elements are initialized to NULL. All elements that are not NULL are queried and are used to calculate the irradiance if the valid neighborhood criterium is satisfied.

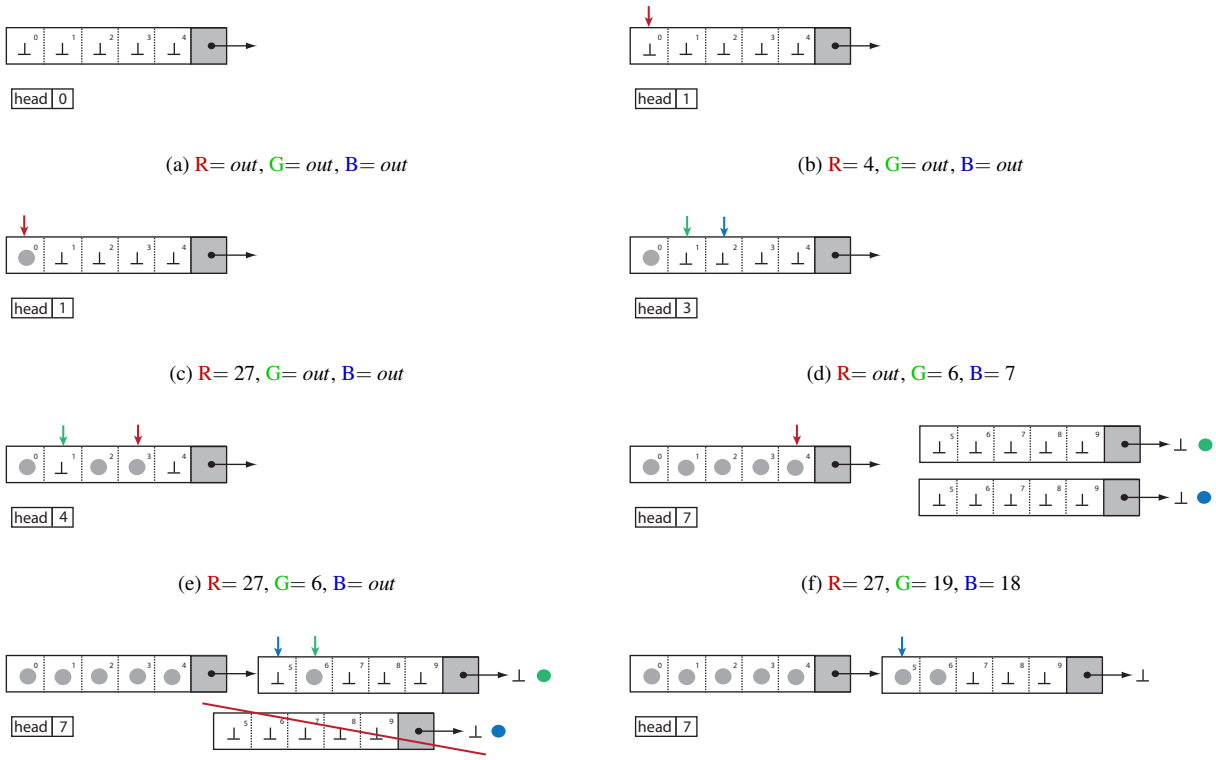
Listing 8: Wait-Free IC getIrradiance

```
1   IrradianceCache::getIrradiance(Irr) {
2       Irr = {0,0,0};
3       <Traverse the octree>
4         for (qNode ∗ tNode = qHead; tNode != NULL; tNode =
              tNode−>next) {
5           for (i = 0; i < qNodeSize; i++)
6               if (tNode−>records[i] != NULL) {
7                   <verify validity of sample>
8                   <extrapoloate irradiance; add to Irr>
9               }
10      }
11      if (found) return true;
12      else return false;
13  }
```

The wait-free approach ensures that the single shared IC can be accessed concurrently by all threads. As we shall show in the next section, this results in faster execution times both when interpolating and creating IC samples and also it does not suffer the larger memory requirements associated with the LW approach.

6. Results

All results presented in this section were obtained on two systems which currently represent the state of the art in multicore technology. One system is a dual quad-core machine based on the Intel Xeon E5520 (Nehalem architecture), running at 2.26 GHz with 12 gigabytes of RAM. These processors include the Intel QuickPath Interconnect, replacing the



(a) R= *out*, G= *out*, B= *out*

(b) R= 4, G= *out*, B= *out*

(c) R= 27, G= *out*, B= *out*

(d) R= *out*, G= 6, B= 7

(e) R= 27, G= 6, B= *out*

(f) R= 27, G= 19, B= 18

(g) R= *out*, G= 27, B= 20

(h) R= *out*, G= *out*, B= 27

Figure 3: An example of how three threads (R, G and B) would concurrently add samples to a node using our novel wait-free method. Numbers refer to the thread's location within Listing 6. *out* entails the thread is not executing this function. ⊥ represents the NULL pointer.

(a) Conference (190k)

(b) Sponza (66k)

(c) Cornell (48k)

(d) Desk (12k)

(e) Office (20k)

Figure 4: The five scenes utilized in the experiments. The polygon count for each scene is shown in brackets.

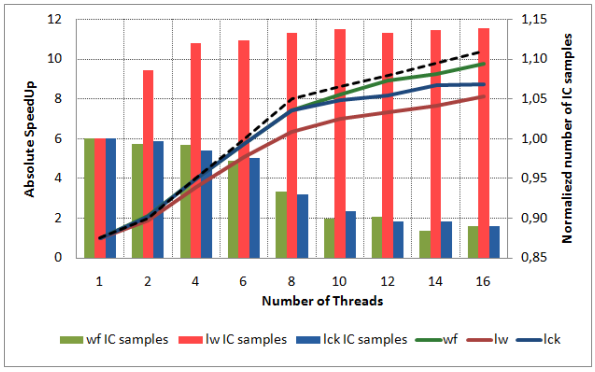
legacy front side bus, and support hyper-threading, enabling two threads per core and thus reporting a total of sixteen logical processors to the operating system. Hyper-threading replicates certain resources of the processor, but not the main execution units; performance is not duplicated, with Intel claiming up to 30% speed improvement, compared to otherwise identical, non hyper-threaded, processors [MBH*02]. The second system is a quad hexa-core machine based on the Intel Xeon E7450 (Dunnington architecture), running at 2.40 GHz with 64 gigabytes of RAM; with a total of 24 physical cores this system enables us to evaluate the scalability of the wait-free access control mechanism proposed in this paper. Both systems run CentOS 5.2, with the code being compiled with Intel Compiler Suite Professional v. 11.0.

For all experiments we used our own interactive ray tracer, which does not make use of packetisation or explicit SIMD operations. The only exception is the ray-bounding volume intersection test used to traverse the acceleration data-structure, which is a BVH implementation based on [WBS07]. Five different scenes (Figure 4) were utilized in the experiments. These scenes were picked to provide a range of geometric complexity, physical dimensions and lighting conditions. All scenes were rendered at a resolution of 640×480. We use the following labels for the methods: traditional sequential method (TRA), lock (LCK), local-write (LW) and wait free (WF).

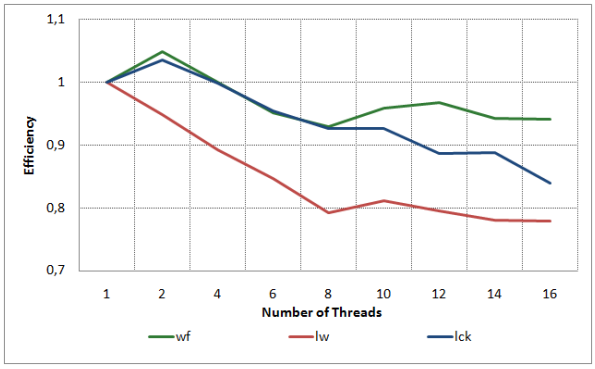
6.1. Still images

Results are presented by varying the number of threads, and thus the number of used cores, up to 16 for the Nehalem architecture and 24 for the Dunnington architecture. The results for a single thread were obtained using TRA, with no access data control, and speed-up for the different techniques is computed with respect to the sequential timings. Each image was calculated with an empty IC to show a worst-case scenario with maximal irradiance calculations occurring.

Figures 5 and 6 present, in graphical form, the speed-up, the normalized number of evaluated IC samples and efficiency, for both architectures. Each of the presented metrics is the average over the five different scenes used in experiments. Since all reported metrics are normalized with respect to the results obtained for the same scene with one single thread and the traditional approach to the IC (no data access control), absolute values, particular to each scene, are not relevant. The later proposition is true as long as behaviour is similar across the different scenes for each access control mechanism. We measured worst-cases standard deviations of 14.2% and 4.2% for absolute speedup and normalized number of generated IC samples, respectively; these low worst-case values indicate that we can use these averages as reliable statistics to analyse our results. For absolute speedup we include a dashed line, depicting linear speedup, which would be obtainable if no algorithmic or implementation penalties were incurred; for the Nehalem architecture above 8 threads, linear speed-up increases only 30% with each additional logical core, following Intel's claim that hyper-threading can provide a maximum 30% speed improvement [MBH*02].



(a) Speedup (lines and left axis) and normalized number of IC samples (bars and right axis)

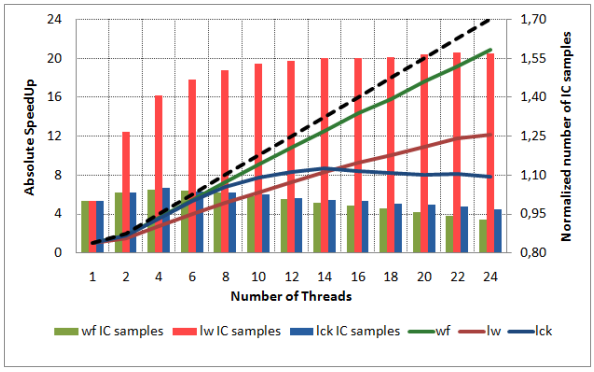


(b) Efficiency

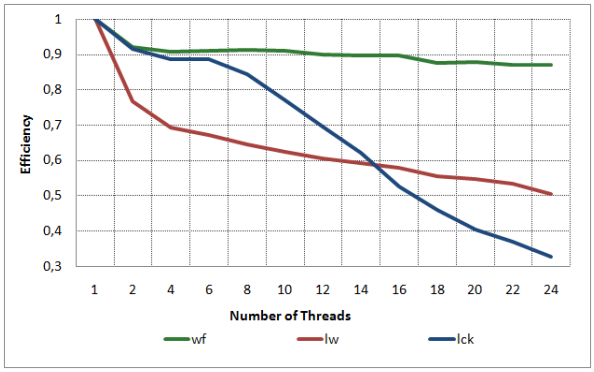
Figure 5: Still Images: results for the Nehalem architecture. (All values are averaged over the 5 different scenes used in experiments)

For all experiments with the Nehalem system, and up to 14 threads in the Dunnington case, LW performs and scales worse than the two other algorithms. This is because no sharing is actually occurring since only one frame is rendered and merging of the local caches only happens at the end of the frame. Each thread must evaluate all irradiance samples that project into its assigned tiles of the image plane, leading to work replication. This can be seen by the number of evaluated irradiance samples (see Figure 5a and Figure 6a), which increases dramatically with the level of concurrency.

The performance difference between LCK and WF becomes evident as the number of threads increases: time waiting for locks grows, resulting in a major performance loss. The wait-free algorithm scales much better. For a reduced number of threads LCK performs similar to WF since most of the time is spent evaluating new irradiance samples, which is not a critical region of the code. As the number of threads



(a) Speedup (lines and left axis) and normalized number of IC samples (bars and right axis)



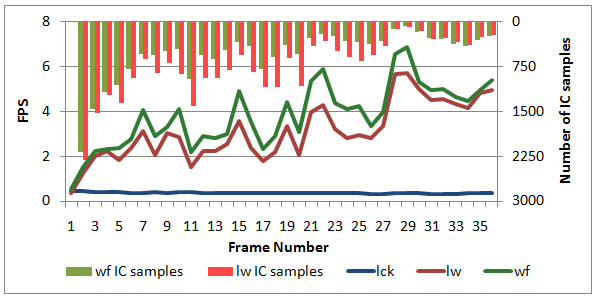
(b) Efficiency

Figure 6: Still Images: results for the Dunnington architecture. (All values are averaged over the 5 different scenes used in experiments)

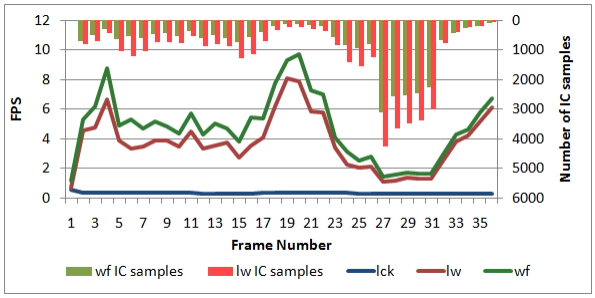
increases, more range searches are performed; since these are serialized in LCK, a performance penalty is incurred. Figures 5a and 6a clearly show that the performance loss incurred by LCK is not due to work replication; in fact, the total number of IC samples evaluated by WF and LCK decreases above a certain number of threads. Success in finding valid samples to interpolate from depends on the order upon which samples are requested and evaluated; concurrent rendering of multiple image plane tiles results in quickly filling the IC with samples that are better distributed over object space thus resulting in more successful range searches than with the sequential approach. Above a significant number of threads, LCK's serialization penalty becomes larger than the overhead associated with work replication and it performs even worse than LW (figure 6).

Parallel algorithms seldom exhibit linear speedup due to overheads, such as load imbalance, work replication and communication/synchronization costs. The wait-free access control mechanism to a shared IC is able to minimize the last two overheads thus exhibiting almost linear speedup and, consequently, a nearly constant efficiency of 0.9 on systems up to 24 cores (see figures 5b and 6b). With around 14 cores on the Dunnington system, LCK speedup reaches an inflection point and starts decreasing, showing that lock-based approaches do not scale with increasing levels of concurrency. On the other hand, WF speedup grows linearly up to 24 cores, although at a rate slightly lower than the increase on the number of cores; the shape of the WF curve suggests that any eventual inflection point is still far from being reached, which demonstrates its superior scalability potential. Note that the shared memory parallel ray tracer incurs additional overheads, such as workload distribution and gathering of results, which are themselves also partially responsible for the small deviation from linear speedup observed with the WF approach.

6.2. Animations



(a) Sponza



(b) Conference Room

Figure 7: Animation Results: Dunnington system with 24 cores. Frames per second (lines and left axis) and number of IC samples per frame (bars and right axis)

Figure 7 shows, for the Sponza and Conference Room scenes running on 24 cores on the Dunnington system, the frame rate (fps) and the number of IC samples evaluated per frame when running an animation of 36 frames while the camera performed a 360 degrees rotation around the scene (10 degrees from frame to frame). Each frame in the sequence re-utilized previously created cache samples while simultaneously calculating new ones. This provides an overview of performance when a mix of evaluation and interpolation is occurring, unlike the case for the still images. For each of the scenes the first frame is the equivalent of the

still images above, where the cache is totally empty and all the samples needed to be generated.

Clearly, LCK performs worse than LW and WF. Since for all frames (except the first) the IC will not be empty, many irradiance samples can be reused, but LCK serializes all range searches performed to locate these samples, thus severely impacting on performance. In fact, with LCK the best rendering time is achieved for the first frame, suggesting that temporal re-utilization of previously calculated irradiance samples is worse than recalculating these values, which completely contradicts the rationale behind the IC [War94]. We can thus conclude that synchronization overheads make the utilization of such lock-based access mechanisms prohibitive when rendering animations of static scenes on highly concurrent shared memory systems.

WF outperforms LW because the former shares irradiance samples immediately without any extra synchronization overhead associated with reading, while the latter does not share samples within a frame, thus resulting in costly extensive evaluations of more indirect diffuse irradiance values. The bars in Figure 7 clearly show that, for WF and LW, variations in performance from frame to frame are highly correlated with the number of IC samples evaluated per frame. More importantly, these graphs also show that the better results achieved with WF are due to evaluating less irradiance samples, which is a consequence of efficient sharing of data among threads.

In summary, LCK is mostly penalized by reading serialization, LW is penalized by work replication, whereas the wait-free approach efficiently shares IC values while minimizing writing overheads and eliminating synchronization overheads associated with concurrent reads.

7. Conclusions

We have presented a wait-free data access control mechanism for sharing the IC among multiple rendering threads on a shared memory parallel system and evaluated it against two traditional data access algorithms: a lock-based approach and a local write one. We demonstrated that the proposed approach outperforms the others and scales better with the number of threads.

The lock-based algorithm serializes all accesses to the shared data structure, reads included. Range searches performed in the octree to locate valid irradiance samples are serialized, resulting in performance losses; this problem is aggravated with the number of threads and the resulting contention. The local write algorithm does not share any irradiance values evaluated within each frame, thus suffering a performance penalty as a result of work replication. Neither of these two algorithms scales well as the number of threads increases.

The wait-free algorithm does not require any critical sections to the shared data structure and the irradiance values are immediately shared among all threads without any synchronization overhead associated with reading. By minimizing the synchronization and work replication overheads which usually plague parallel systems, the wait-free algorithm exhibits the best rendering times for both still images and walkthroughs within static scenes and scales well with the number of threads, achieving a near linear speedup for up to 24 threads.

Multicore systems now represent the standard form of desktop computing. Since in the near future such systems will have a degree of parallelism which is expected to be larger than that on current machines, the relevance of efficient, scalable and reliable shared data structures for maximizing performance is ever increasing. Alternatives to the status quo of locking and blocking in the form of wait-free data structures can offer a number of advantages. These methods can make it possible for traditional graphics algorithms to exploit modern hardware. In this paper we have demonstrated the potential of such techniques in the form of a shared memory IC. Our algorithm has made it possible to achieve close to interactive rates for ray tracing with global illumination. We hope that our solution will motivate similar parallel methods in other areas of computer graphics.

8. Acknowledgement

This research was partially funded by project IGIDE, PTDC/EIA/65965/2006 funded by the Portuguese Foundation for Science and Technology, and UK-EPSRC grant EP/D069874/2. We thank Greg Ward for the Office and Conference scenes from the Radiance package and the Stanford's Graphics Group for the Bunny model from the Stanford 3D Repository. Finally, we thank Alberto Proença for granting us access to the two multicore systems.

References

[DDSC09] DUBLA P., DEBATTISTA K., SANTOS L. P., CHALMERS A.: Wait-Free Shared-Memory Irradiance Cache. Debattista K., Weiskopf D., Comba J., (Eds.), Eurographics Association, pp. 57–64.

[DSC06] DEBATTISTA K., SANTOS L. P., CHALMERS A.: Accelerating the irradiance cache through parallel component-based rendering. In *Eurographics Symp. on Parallel Graphics and Visualization* (2006).

[Her09] HERLIHY M.: Technical perspective highly concurrent data structures. *Commun. ACM 52*, 5 (2009), 99–99.

[HLM03] HERLIHY M., LUCHANGCO V., MOIR M.: Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems* (Washington, DC, USA, 2003), IEEE Computer Society, p. 522.

[HW90] HERLIHY M. P., WING J. M.: Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (1990), 463–492.

[KMG99] KOHOLKA R., MAYER H., GOLLER A.: MPI-parallelized Radiance on SGI CoW and SMP. In *ParNum'99: 4th Int. ACPC Conf.* (1999), pp. 549–558.

[MBH*02] MARR D., BINNS F., HILL D., HINTON G., KOUFATY D., MILLER A., UPTON M.: Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal 6*, 1 (February 2002), 4–15.

[RCLL99] ROBERTSON D., CAMPBELL K., LAU S., LIGOCKI T.: Parallelization of radiance for real time interactive lighting visualization walkthroughs. In *ACM/IEEE Supercomputing* (1999), ACM Press, p. 61.

[War88] WARD G.: A ray tracing solution for diffuse interreflection. *Computer Graphics - SIGGRAPH'88 22*, 4 (August 1988).

[War94] WARD G.: The radiance lighting simulation and rendering. *Computer Graphics - SIGGRAPH'94* (1994).

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26*, 1 (2007), 6.

[WMG*07] WALD I., MARK W. R., GÜNTHER J., BOULOS S., IZE T., HUNT W., PARKER S. G., SHIRLEY P.: State of the art in ray tracing animated scenes. In *STAR Proceedings of Eurographics 2007* (September 2007), Schmalstieg D., Bittner J., (Eds.), pp. 89–116.

Appendix A: Tables of results

Table 1 and Table 2 show the timing, speedup results and IC samples for each of the IC methods for the results on the 8-core and 24-core respectively. The results are shown for each of the scenes in Figure 4.

	1	**2**			**4**			**8**			**12**			**16**		
	TRA	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*
Cornell																
Time (s)	9.33	4.14	4.36	4.10	2.40	2.54	2.27	1.40	1.67	1.43	1.47	1.54	1.26	1.41	1.40	1.15
IC samples	4885	4939	5286	4916	4854	5427	4863	4406	5471	4416	4196	5454	4198	4220	5472	4171
Speed-up	1.00	2.25	2.14	2.28	3.88	3.67	4.12	6.67	5.60	6.53	6.45	6.05	7.41	6.61	6.68	8.14
Desk																
Time (s)	8.96	4.20	4.72	4.50	2.24	2.57	2.31	1.21	1.47	1.23	1.04	1.20	1.00	1.02	1.10	0.89
IC samples	3953	3957	4426	3916	3914	4520	3955	3684	4552	3751	3538	4577	3581	3444	4591	3516
Speed-up	1.00	2.13	1.90	1.99	4.00	3.48	3.88	7.38	6.11	7.30	8.64	7.47	8.98	8.81	8.17	10.04
Conf.																
Time (s)	16.46	9.26	10.10	8.71	4.57	5.25	4.61	2.37	2.79	2.36	2.18	2.64	2.02	1.95	2.27	1.91
IC samples	5378	5345	5842	5310	5319	6046	5278	5159	6207	5120	4867	6185	4876	4837	6207	4790
Speed-up	1.00	1.78	1.63	1.89	3.60	3.13	3.57	6.96	5.91	6.98	7.56	6.74	8.15	8.44	7.25	8.63
Office																
Time (s)	8.63	3.95	4.53	3.96	1.96	2.29	1.98	1.05	1.26	1.02	0.96	1.06	0.85	0.92	0.92	0.75
IC samples	3910	3808	4253	3835	3738	4523	3847	3495	4518	3537	3388	4505	3431	3370	4566	3351
Speed-up	1.00	2.18	1.90	2.18	4.41	3.77	4.37	8.19	6.86	8.43	9.00	8.18	10.10	9.39	9.38	11.58
Sponza																
Time (s)	36.06	18.05	18.81	16.87	8.84	9.55	8.92	4.58	5.01	4.55	3.95	4.44	3.65	3.48	3.99	3.43
IC samples	8196	8232	8619	8214	8190	8723	8209	7907	8893	7863	7835	8918	7885	7802	8932	7843
Speed-up	1.00	2.00	1.92	2.14	4.08	3.78	4.04	7.87	7.19	7.92	9.13	8.13	9.88	10.37	9.05	10.52

Table 1: Results for still images on the 8-core Nehalem architecture. Note that speedup calculations were calculated on timing results of up to five decimal places (as opposed to the two decimal places shown here).

	1	**2**			**4**			**8**			**16**			**24**		
	TRA	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*	*LCK*	*LW*	*WF*
Cornell																
Time (s)	9.17	4.78	5.55	4.76	2.46	3.07	2.39	1.31	1.69	1.23	1.29	1.06	0.70	2.03	0.91	0.50
IC samples	4888	5018	6027	5048	5069	6683	5040	4939	7106	4963	4820	7320	4803	4677	7353	4323
Speed-up	1.00	1.92	1.65	1.92	3.74	2.99	3.84	7.02	5.43	7.48	7.11	8.64	13.02	4.52	10.11	18.38
Desk																
Time (s)	8.71	4.92	6.10	4.88	2.45	3.23	2.42	1.29	1.76	1.19	1.78	0.97	0.59	2.31	0.78	0.40
IC samples	3849	3964	5056	3939	4055	5629	4088	3984	5974	3987	3821	6101	3677	3664	6121	3349
Speed-up	1.00	1.77	1.43	1.79	3.55	2.69	3.59	6.72	4.95	7.30	4.88	8.97	14.82	3.77	11.16	21.69
Conf																
Time (s)	18.14	9.92	12.22	9.87	5.26	6.94	5.22	2.62	3.68	2.59	1.55	1.95	1.24	1.77	1.40	0.83
IC samples	5153	5315	6651	5330	5464	7391	5457	5427	7965	5429	5225	8218	5114	5103	8384	4982
Speed-up	1.00	1.83	1.49	1.84	3.45	2.61	3.48	6.91	4.93	7.01	11.71	9.32	14.60	10.27	12.99	21.73
Office																
Time (s)	8.31	4.92	5.73	4.74	2.51	3.20	2.39	1.43	1.66	1.15	1.57	0.89	0.56	1.93	0.69	0.39
IC samples	3770	4067	4915	3955	4033	5540	3955	3936	5947	3985	3883	6149	3749	3744	6179	3654
Speed-up	1.00	1.69	1.45	1.75	3.31	2.59	3.48	5.80	5.00	7.25	5.31	9.37	14.79	4.31	12.07	21.17
Sponza																
Time (s)	39.22	20.06	23.49	20.32	10.65	13.15	10.40	5.39	7.09	5.23	2.98	3.93	2.69	2.39	2.75	1.81
IC samples	8005	8043	9489	8169	8249	10356	8073	8161	11063	8005	7802	11615	7897	7660	11795	7654
Speed-up	1.00	1.96	1.67	1.93	3.68	2.98	3.77	7.28	5.53	7.50	13.16	9.99	14.56	16.40	14.28	21.61

Table 2: Results for still images on the 24-core Dunnington architecture. Note that speedup calculations were calculated on timing results of up to five decimal places (as opposed to the two decimal places shown here).